The progress of session types

Vasco T. Vasconcelos, University of Lisbon

25th International Conference on Concurrency Theory, Rome, 4 September 2014
A bit of history

• The year is 1989, the year of the first edition of CONCUR.

• Exciting news arrive from Europe…
A Calculus of Mobile Processes,
Part I

by

Robin Milner
Joachim Parrow
David Walker
Rapports de Recherche

N° 1154

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

FUNCTIONS AS PROCESSES

Robin MILNER

Février 1990
Sorts and Types in the \( \Pi \)-calculus

Robin Milner

December 1990

1. Introduction
2. Abstraction, objects and sorts
3. Examples of sortings
4. Types, with elementary examples
5. Types of \( \lambda \)-agents
6. Higher-order?

APPENDIX: Structural congruence, actions, and transition rules
Kohei Honda _ 1959-2012

Kohei was then a student at Keio University, Japan

Geneva, 1991
Every computational behaviour can be reduced to name passing
Types for Dyadic Interaction*

Kohei Honda
kohei@ml.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We formulate a typed formalism for concurrency where types denote freely composable structure of dyadic interaction in the symmetric scheme. The resulting calculus is a typed reconstruction of name passing process calculi. Systems with both the explicit and implicit typing disciplines, where types form a simple hierarchy of types, are presented, which are proved to be in accordance with each other. A typed variant of bisimilarity is formulated and it is shown that typed $\beta$-equality has a clean embedding in the bisimilarity. Name reference structure induced by the simple hierarchy of types is studied, which fully characterises the typable terms in the set of untyped terms. It turns out that the name reference structure results in the deadlock-free property for a subset of terms with a certain regular structure, showing behavioural significance of the simple type discipline.
An Interaction-based Language and its Typing System*

Kaku Takeuchi  Kohei Honda  Makoto Kubo
kaku@mt.cs.keio.ac.jp  kohei@mt.cs.keio.ac.jp  kubo@mt.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We present a small language $\mathcal{L}$ and its typing system based on the idea of interaction, one of the important notions in parallel and distributed computing. $\mathcal{L}$ is based on, apart from such constructs as parallel composition and process creation, three pairs of communication primitives which use the notion of a session, a semantically atomic chain of communication actions which can interleave with other such chains freely, for high-level abstraction of interaction-based computing. Three primitives enable programmers to elegantly describe complex interactions among processes with a rigorous type discipline similar to ML [4]. The language is given formal operational semantics
LANGUAGE PRIMITIVES AND TYPE DISCIPLINE FOR STRUCTURED COMMUNICATION-BASED PROGRAMMING

KOHEI HONDA*, VASCO T. VASCONCELOS†, AND MAKOTO KUBO‡

ABSTRACT. We introduce basic language constructs and a type discipline as a foundation of structured communication-based concurrent programming. The constructs, which are easily translatable into the summation-less asynchronous π-calculus, allow programmers to organise programs as a combination of multiple flows of (possibly unbounded) reciprocal interactions in a simple and elegant way, subsuming the preceding communication primitives such as method invocation and rendez-vous. The resulting syntactic structure is exploited by a type discipline à la ML, which offers a high-level type abstraction of interactive behaviours of programs as well as guaranteeing the compatibility of interaction patterns between processes in a well-typed program. After presenting the formal semantics, the use of language constructs is illustrated through examples, and the basic syntactic results of the type discipline are established. Implementation concerns are also addressed.
Sessions are

“...structuring constructs for communication-based programming which are born through the examination of pi-calculus encoding of various computational structures”
A session is ...

“...a semantically atomic chain of communication actions which can interleave with other such chains freely, for high-level abstraction of interaction-based computing.”
Session types

\[ T, U ::= \text{end} \quad \text{no further interaction} \]
\[ | \quad !T.U \quad \text{output } T, \text{ proceed as } U \]
\[ | \quad ?T.U \quad \text{input } T, \text{ proceed as } U \]
Example _ The type of a binary integer predicate

!int.!int.?bool.end as seen from the “client” side

?int.?int.!bool.end as seen from the “server” side
Duality _ A fundamental notion

\[
\begin{align*}
T_1 &= T_2 \quad U_1 \text{ dualof } U_2 \\
!T_1.U_1 &= \text{ dualof } ?T_2.U_2
\end{align*}
\]

\[
\begin{align*}
T_1 &= T_2 \quad U_1 \text{ dualof } U_2 \\
?T_1.U_1 &= \text{ dualof } !T_2.U_2
\end{align*}
\]
Choice

* Alternatives:

1. Binary choice

2. Labelled (n-ary) choice _ goes better with programming languages

\[ T, U ::= \ldots \]

\[ | & \{l_1: T_1, \ldots, l_n: T_n\} \quad \text{offer a } l_1, \ldots, l_n \text{ choice;} \]

\[ | \oplus \{l_1: T_1, \ldots, l_n: T_n\} \quad \text{if } l_i \text{ is picked, proceed as } T_i \]

\[ \text{select option } l_j; \]

\[ \text{if } l_i \text{ is picked, proceed as } T_j \]
Duality for choice

\[
\begin{align*}
T_1 \text{ dualof } U_1 & \ldots & T_n \text{ dualof } U_n \\
\& \{l_1 : T_1, \ldots, l_n : T_n\} \text{ dualof } \ominus \{l_1 : U_1, \ldots, l_n : U_n\}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
T_1 \text{ dualof } U_1 \\
\oplus \{l_1 : T_1, \ldots, l_n : T_n\}
\end{array} & \ldots & \begin{array}{c}
T_n \text{ dualof } U_n \\
\& \{l_1 : U_1, \ldots, l_n : U_n\}
\end{array}
\end{align*}
\]
What can session types model?

- Interactions on some communication medium:
  1. Object references
  2. Communication channels
  3. ...
interface Iterator {
    boolean hasNext ();
    Object next ();
    void remove ();
}
Can you spot the flaws?

```java
void commaSeparatedList (Iterator it) {
    System.out.print(it.next());
    while (it.hasNext())
        System.out.print(",", " + it.next()); }

void filter (Iterator it, Object o) {
    while (it.hasNext())
        if (it.next().equals(o))
            System.out.print(it.next()); }

void removeFirst (Iterator it) {
    if (it.hasNext())
        it.remove(); }
```
This code...

- Compiles

- and may even run...

- but runs may as well end up with a NoSuchElementException or a IllegalStateException

- To “correctly” use the iterator one must read the documentation
public interface Iterator<E> {
    /**
     * Returns <tt>true</tt> if the iteration has more elements. (In other
     * words, returns <tt>true</tt> if <tt>next</tt> would return an element
     * rather than throwing an exception.)
     *
     * @return <tt>true</tt> if the iterator has more elements.
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration. Calling this method
     * repeatedly until the {@link #hasNext()} method returns false will
     * return each element in the underlying collection exactly once.
     *
     * @return the next element in the iteration
     * @exception NoSuchElementException iteration has no more elements.
     */
    E next();

    /**
     * Removes from the underlying collection the last element returned by the
     * iterator (optional operation). This method can be called only once per
     * call to <tt>next</tt>. The behavior of an iterator is unspecified if
     * the underlying collection is modified while the iteration is in
     * progress in any way other than by calling this method.
     *
     * @exception UnsupportedOperationException if the <tt>remove</tt>
     * operation is not supported by this Iterator.
     * @exception IllegalStateException if the <tt>next</tt> method has not
     * yet been called, or the <tt>remove</tt> method has already
     * been called after the last call to the <tt>next</tt>
     * method.
     */
    void remove();
}
A session type for the iterator as seen from the point of view of the client

\[
\oplus \{ \text{hasNext: } \&\{ \text{true: } \oplus \{ \text{next: } \oplus \{ \text{remove: } \ldots
\text{hasNext: } \ldots \} \\}
\text{hasNext: } \ldots \} \}
\text{false: end} \}
\]

- as seen from the point of view of the client
- (Gay, Vasconcelos, et al., POPL’10)
The technical ingredients

- We work with the pi-calculus
Typing input and output

\[
\frac{\Gamma, x : U \vdash P}{\Gamma, x : !T.U, y : T \vdash x!y.P}
\]

\[
\frac{\Gamma, x : U, y : T \vdash P}{\Gamma, x : ?T.U \vdash x?y.P}
\]

- The type of a channel changes throughout the typing derivation
Reduction

\[(\nu xy)(x!v.P | y?z.Q) \rightarrow (\nu xy)(P | Q[v/z])\]

Syntactically distinguish the two ends of a channel
Parallel composition, channel creation and inaction

\[
\frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma, \Delta \vdash P \mid Q}
\]

\[
\frac{\Gamma, x: T, y: U \vdash P \quad T \text{ dual of } U}{\Gamma \vdash (\nu x: T, y: U)P}
\]

\[
\vdash 0
\]
A word on typing contexts

- Duplicated entries ok, but

  \[
  \text{If } x : T \text{ and } x : U \text{ in } \Gamma \text{ then } T = U = \text{end}
  \]

- Structural rules _ Weakening and Contraction:

  \[
  \frac{\Gamma \vdash P}{\Gamma, x : \text{end} \vdash P}
  \]

  \[
  \frac{\Gamma, x : \text{end}, x : \text{end} \vdash P}{\Gamma, x : \text{end} \vdash P}
  \]
Examples

\[ \text{type } T = !\text{end} . \text{end} \]

\[ \text{type } U = ?\text{end} . \text{end} \]

\[ u : \text{end} \vdash (\nu x : T, y : U)(x!u . 0 \mid y?z . 0) \]

\[ u : \text{end} \vdash (\nu a : !T . \text{end}, b : ?T . \text{end})(\nu x : T, y : U)(a!x \mid b?w . (w!u \mid y?z)) \]

\[ u : \text{end} \vdash (\nu x : !\text{end}. U, y : ?\text{end}. T)(x!u . x?z \mid y?w . y!u) \]

\[ u : \text{end} \not\vdash (\nu x : T, y : U)(x!u \mid x!u \mid y?z) \]

\[ u : \text{end} \not\vdash (\nu x : T, y : U)(y?z) \]

- \text{(linear) channel passing}
- \text{shared}
- \text{linear}
- \text{Race condition}
- \text{Unconsumed channel end}
Recursive types

- Standard mu types
- Type equality co-inductive
- Duality co-inductive
- One more structural rule allowing replacing equals by equals in typing contexts
The shared world

- There’s very little one can do with linear channels alone (we sometimes want races)

- Alternatives:

  1. Superimpose the shared world: different syntactic categories for linear/shared types and channels, different syntax for linear/shared message passing (Honda, Vasconcelos, Kubo, ESOP’98)

  2. Label types with lin/un qualifiers (Vasconcelos, I&C’12)
Types revisited

\[
T, U ::= \text{end} \\
\quad | \ q!T.U \\
\quad | \ q?T.U \\
\quad | \ \mu a.T \\
\quad | \ a \\
q ::= \text{lin} \\
\quad | \ \text{un}
\]
Type formation and the un predicate

- Type formation:

  \[ \text{un}!T.U \text{ is well formed if } \text{un}!T.U = U \]

- The un predicate:

  \[
  \begin{array}{c}
  \text{end un} \\
  \text{un}!T.U \text{ un} \\
  \mu_a.T \text{ un}
  \end{array}
  \]

- Allow duplicated un types in contexts
Typing the shared world

\[
\begin{align*}
\Gamma, x : U &\vdash P \\
\Gamma, x : q!T.U, y : T &\vdash x!y.P \\
\Gamma, x : U, y : T &\vdash P \\
\Gamma, x : q?T.U &\vdash x?y.P
\end{align*}
\]

\[\text{type } T = \mu a. \text{un!end}.a\]
\[\text{type } U = \mu b. \text{un?end}.b\]
\[u : \text{end} \vdash (\nu x : T, y : U)(x!u \mid x!u \mid y?z)\]
\[u : \text{end} \vdash (\nu x : T, y : U)(y?z)\]

races ok
unconsumed channels ok
Unbounded behaviour

- Up until now all computations terminate
- Alternatives:
  1. Equations (Milner, Parrow, Walker, I&C’92)
  2. Rec process constructors (Honda, Vasconcelos, Kubo, ESOP’98)
  3. Replication (Milner, MSCS’92; Gay, Hole, Acta Informatica’05)
Replication

\[(\nu xy)(x!v.P \mid y*?z.Q) \rightarrow (\nu xy)(P \mid Q[v/z] \mid y*?z.Q)\]

\[
\frac{
\Gamma, x: U, y: T \vdash P \quad \Gamma \text{ un}
}{
\Gamma, x: \text{un} ? T.U \vdash x*?y.P
}\]
Example _ Interaction with a shared server (session initiation)

(νclient, server)

(νx, y)
server!x.
y!5.y?z.print!z

| client*?x.
x?w.x!(w + 1)

Client code
create a fresh session channel
send one end to the server
interact on the other end

Server code
get a session end from the client
read and increment
Extended typability _ Types that evolve from lin to un

\[
\text{type } T = \text{lin}!\text{bool.} \mu a. \text{un}!\text{int.a}
\]

\[
\text{type } U = \text{lin}?\text{bool.} \mu b. \text{un}!\text{int.b}
\]

\[
\vdash (\forall x : T, y : U)(x!\text{true}.(x!5.x!7 \mid x!9) \mid y?z.y*?w)
\]

✝ Not typable in systems that use different syntactic categories for shared and linear types

✝ Evolving from un to lin is unsound
Main results

- Soundness _ Typability preserved by reduction
- Fidelity _ Code exchanges messages as prescribed by the types
- Type equality and duality decidable
- Type checking decidable (and polynomial) _ Algorithmic type checking
Challenges I _ Progress (absence of deadlocks)

- An easy deadlock:
  \[
  \vdash (\forall x, y)(\forall a, b) (x!5.a!true \mid b?w.y?z)
  \]

- Possible ways out:
  1. Setup a partial order on channel names
  2. Setup a p.o. on actions (!, ?, &, +) in session types
  3. Allow one open session per thread
  4. Multiparty session types
  5. Use a non session-based type system
Challenges II _ Extending applicability

✤ Can we do “session types” for

✤ Actor systems _ one reader, many writers

✤ Parallel programs

✤ Distributed (consensus) algorithms _ processes play different roles, state change based on the number of messages received

✤ ...

Further applications

- Statically check code against session types
- Monitor unchecked applications against ST
- Synthesise (the communication part of) code from ST
- Test code against ST _ integration tests
Conclusion

- 21 years of session types
- Rich theory, tight connection to linear logic
- Wide applicability
You may want to have fun with SePi


- A pi-calculus based language with (linearly refined) session types